Session Code: TLS310

tools & languages

Whidbey

Visual C++ “Whidbey”:
New Language Design And Enhancements

Herb Sutter
Architect, Microsoft Visual C++
hsutter@microsoft.com

PDC03

Make the connection

Quake II Takeaways

960x720 + software-rendered on 1.2GHz PIII-M + Fx 1.1.4322

1. It's easy to port C/C++ code to .NET: 100% JITted (IL) code; still native data
   - Just rebuild with /clr
   - **1 day** to port the entire Quake 2 source base. (Nearly all of the effort was to translate from C to C++, and had nothing to do with our compiler or the .NET platform)
2. It's not hard to extend existing code with CLR types
   - **2 days** to implement the radar extension using Fx (gradient brushes, window transparency/opacity, Matrix.RotateAt)
3. It needs to be still easier, more natural,

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Rationale

- C++: First-class .NET development language
  - Remove “Why Can’t I” usability and migration barriers:

    Port and extend existing programs even more seamlessly
  - Key Q: “Why should a .NET developer use C++?”
  - Deliver promise of **C**LR
- "Managed C++" doesn’t get us there:
  - Great for basic interop, migrating existing code to .NET
  - Second-class CLR support (e.g., __property)
  - Poor integration of C++ and CLR features and idioms (e.g., no templates of CLR types)
  - Hard to write pure (verifiable, secure) .NET apps
  - Ugly and nonintuitive syntax, uneven and contorted semantics. Failed to achieve a natural, organic, “everything in its place” surfacing of features

Major Goals

- Feature coverage:
  - Provide organic first-class support for CLR features/idioms
    - Example: Verifiability at first try in this complete program:

```
int main()
{ System::Console::WriteLine( "Hello, world!" ); }
```

  - Leave no room for a language lower than C++ (incl. IL)
- Usability and adoptability:
  - More elegant syntax, natural & pure extensions to ISO C++
- C++ × CLR:
  - "Bring C++ to .NET": Support C++'s powerful features also for CLR types (e.g., deterministic

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Basic Class Declaration Syntax

- Type are declared "*adjective* class":

```
class N{ /*...*/ };  // native
ref class R{ /*...*/ };  // CLR reference type
value class V{ /*...*/ };  // CLR value type
interface class I        { /*...*/ };    // CLR interface type
enum class E{ /*...*/ };  // CLR enumeration type
```

  - C++ and .NET fundamental types are mapped to each other (e.g., int and System::Int32 are the same type)
- Any type can:
  - Have a destructor ~T(), and/or finalizer !T()
  - Have a copy constructor, and/or copy assignment operator:
    - Value class always have them. Native classes have them by default. Ref classes do not have them by default

Properties

- Basic syntax:

```
ref class R {
  int mySize;
public:
  property int Size {
    int get()                { return mySize; }
    void set( int val )      { mySize = val; }
  }
};

R r;
r.Size = 42;                 // use like a field; calls
r.Size::set(42)
```

- Trivial properties:

```
ref class R {
public:
  property int Size;         // compiler-generated
};                           // get, set, and backing
```

Indexed Properties

- Indexed syntax:

```
ref class R { // ...
  map<String^,int>* m;
public:
  property int Lookup[ String^ s ] {
    int get()                  { return (*m)[s]; }
  protected:
    void set( int );           // defined out of line below
  }
  property String^ default[ int i ] { /*...*/ }
};
void R::Lookup[ String^ s ]::set( int v ) { (*m)[s] =
v; }
```

- Call point:

```
R r;
r.Lookup["Adams"] = 42;       //
r.Lookup[“Adams”].set(42)
```

Delegates and Events

- A trivial event:

```
delegate void D( int );

ref class R {
public:
  event D^ e;        // trivial event; compiler-generated members
  void f() { e( 42 ); }        // invoke it
};

R r;
r.e += gcnew D( this, &SomeMethod );
r.e += gcnew D( SomeFreeFunction );
r.f();
```

- Or you can write add/remove/raise yourself

Virtual Functions and Overriding

- Explicit, multiple, and renamed overriding:

```
interface class I1 { int f();      int g();                  };
interface class I2 { int f();                int h();        };
interface class I3 {                         int h(); int i(); };
ref class R : I1, I2, I3 {
public:
  virtual int ff() override; // error, there is no virtual ff()
  virtual int f() sealed;// overrides & seals I1::f and I2::f
  virtual int x() = I1::g;    // overrides I1::g
  virtual int y1() = I2::h;  // overrides I2::h
  virtual int z() = i, I3::h; // overrides I3::h and I3::i
  virtual int a() abstract; // same as “= 0” (for symmetry
                                                  with class
```

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Unified Storage/Pointer Model

- Semantically, a C++ program can create object of any type T in any storage location:
  - On the native heap:
    ```
    T* t1 = new T;
    ```
    - As usual, pointers (*) are stable, even during GC
    - As usual, failure to explicitly call delete will leak
  - On the gc heap:
    ```
    T^ t2 = gcnew T;
    ```
    - Handles (^) are object references (to whole objects)
    - Calling delete is optional: "Destroy now, or finalize later."
  - On the stack, or as a class member: `T t3;`
    - Q: Why would you? A: Next section: Deterministic destruction/dispose is automatic and implicit, hooked to stack unwinding or to the enclosing object's lifetime

Pointers

- Native pointers (*) and handles (^):
  - ^ is like *, except ^ points to a whole object on the gc heap, can't be ordered, and can't be cast to void* (or integral type) and back. (There is no void^.)

```
Widget* s1 = new Widget;     // point to native heap
Widget^ s2 = gcnew Widget;     // point to gc heap

s1->Length();                          // use -> for member access
s2->Length();

(*s1).Length();       // use * to dereference
(*s2).Length();
```

- Use RAII pin_ptr to get a * into the gc heap:

```
R^ r = gcnew R;
int* p1 = &r->v;    // error, v is a gc-lvalue
pin_ptr<int> p2 = &r->v;       // ok
```

References And Unary &/%

- Native (&) and tracking (%) references:
  - % is like &, except % can refer into any memory area incl. the gc heap (and for now a % can only exist on the stack)

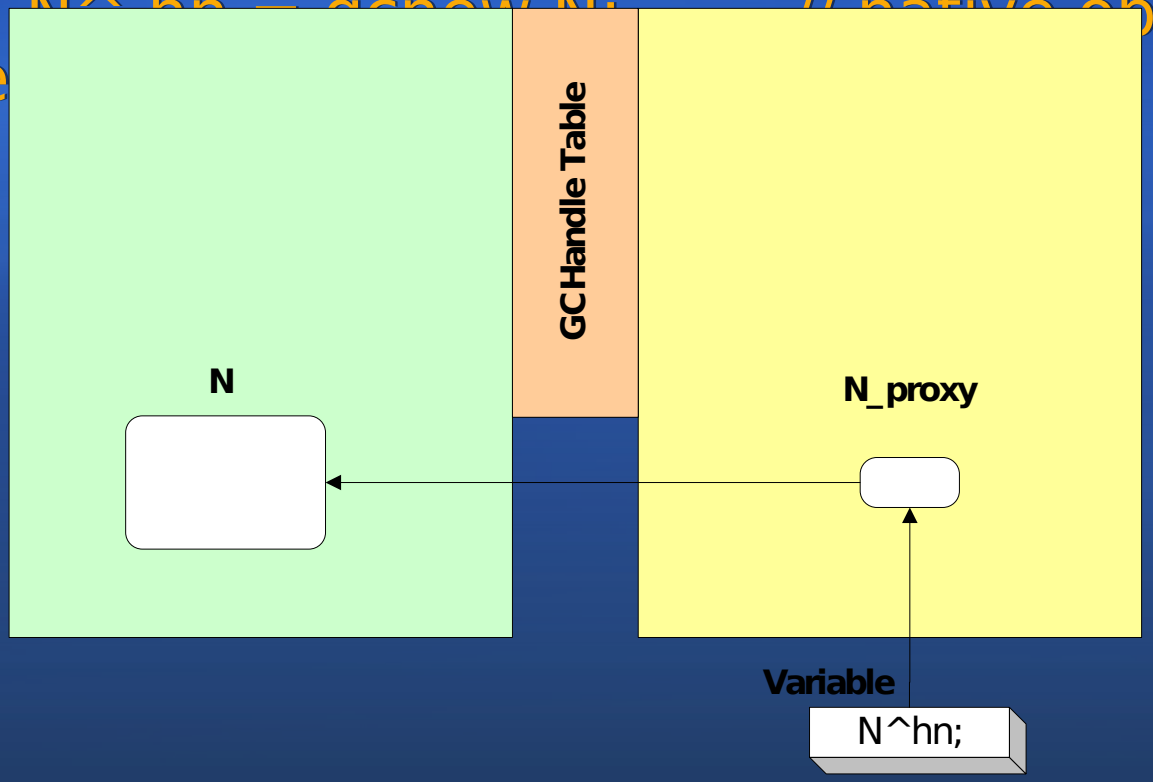
```
String& s3 = *s1;  // bind
String% s4 = *s2;  // bind & track

s3.Length();                              // reference syntax with
s4.Length();

void swap( Object^% o1, Object^% o2 )        // C# “ref”
  { Object^ tmp = o1; o1 = o2; o2 = tmp; }
```

- Unary & and % for “address of”:
  - &myobj → MyType* (or interior_ptr<MyType>, when myobj is physically on the gc heap)

Native On The GC Heap

- Create a proxy for native object on gc heap
  - The proxy's finalizer will call the destructor if needed

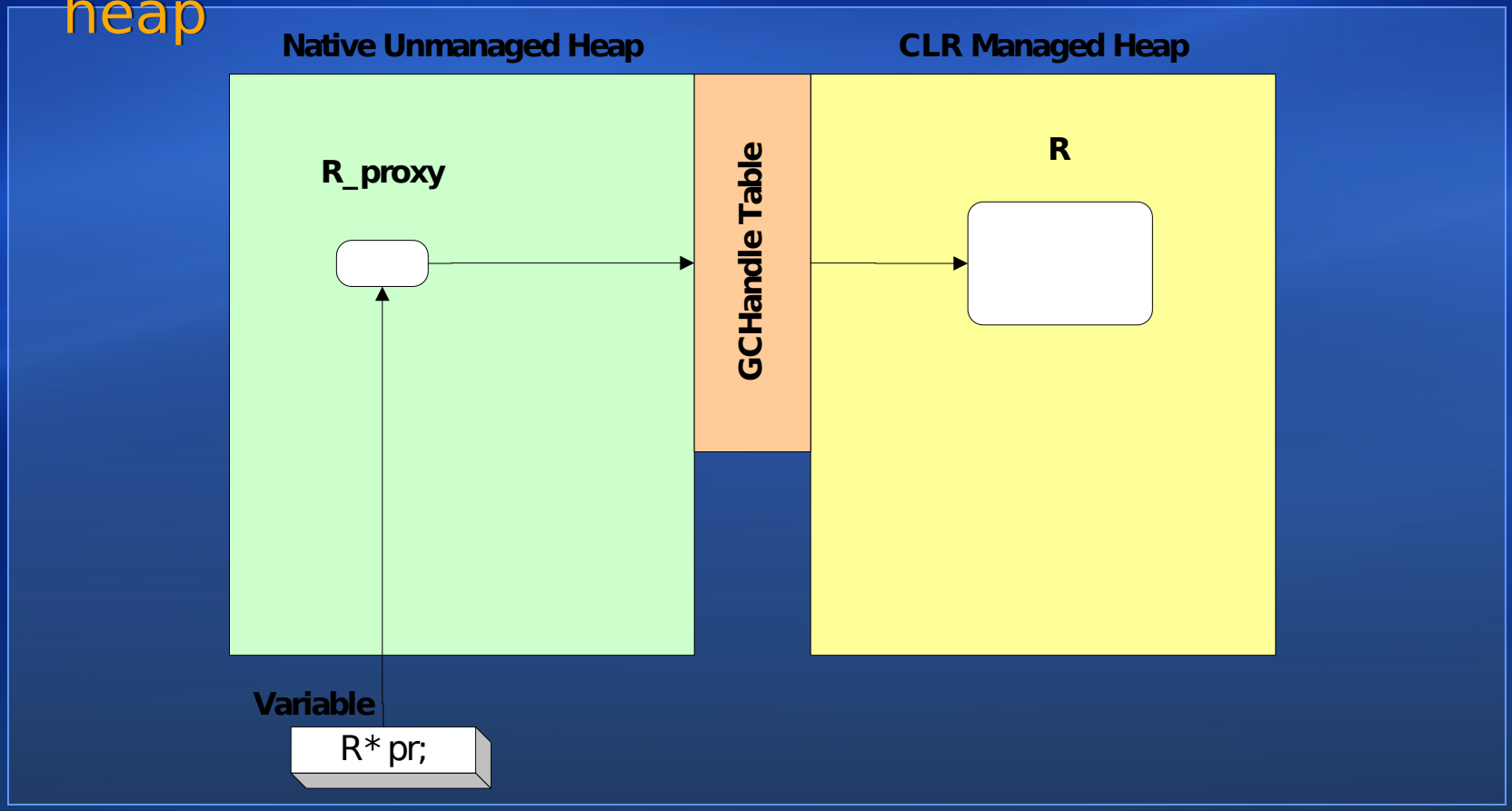
N^ hn = gcnew N; // native object on gc he



Ref Class On Native Heap

- Already implemented as gcroot template
  - No finalizer will ever run. Example:

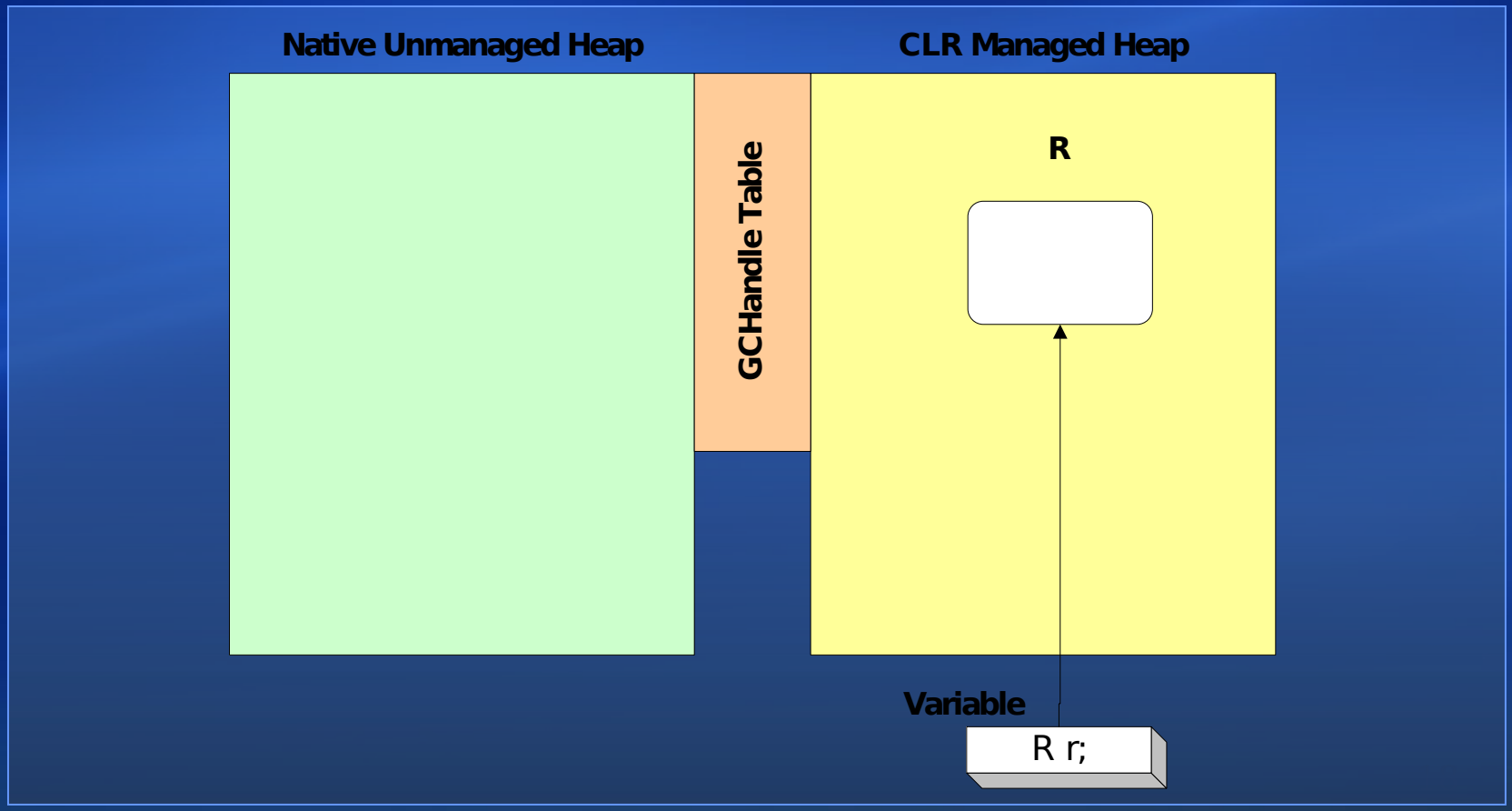
```
R* pr = new R;     // ref object on native heap
```



Ref Class On The Stack

- The type of “%R” is R^.

```
R r;                    // ref object on stack
f( %r );                // call f( Object^ )
```



Boxing (Value Types)

- Boxing is implicit and strongly typed:

```
int^ i = 42;          // strongly typed boxed value
Object^ o = i;        // usual derived-to-base conversions ok

Console::WriteLine( "Two numbers: {0} {1}", i, 101 );
```

  - i is emitted with type Object + attribute marking it as int.
    WriteLine chooses the Object overload as expected
  - Boxing invokes the copy constructor
- Unboxing is explicit:
  - Dereferencing a V^ indicates the value inside the box, and this syntax is also used for unboxing:

```
int k = *i;           // unboxing to take a copy

int% i2 = *i;         // refer into the box (no copy)
```

Aside: Agnostic Templates

- To demonstrate the unification, consider agnostic templates
- Example 1: Usual swap, with % instead of &

```
template<class T>
void swap( T% t1, T% t2 )
  { T tmp( t1 ); t1 = t2; t2 = tmp; }
```

  - Works for any copyable T:

```
Object ^o1, ^o2;  swap( o1, o2 );                 // swap handles
int ^i1, ^i2;                    swap( i1, i2 );  // swap handles
                                 swap( *i1, *i2 ); // swap values
MessageQueue *q1, *q2;           swap( q1, q2 );  // swap pointers
                                 swap( *q1, *q2 );// swap values
```

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Cleanup: Less Code, More Control

- The CLR state of the art is great for memory
- It's not great for other resource types:
  - Having lots of finalizers doesn't scale, and usually the finalizer is run too late. Examples: files, database connections, locks
  - The Dispose pattern (try-finally, or C# "using") tries to address this, but is fragile, error-prone, and requires the user to write more code
- Instead of writing try-finally or using blocks:
  - Users can leverage a destructor. The C++ compiler generates all the Dispose code automatically, including chaining calls to Dispose. (There is no Dispose pattern)
  - Types authored in C++ are naturally usable in other languages, and vice versa
  - C++ choice: Correctness by default, speed by choice. (Other langs: Speed by default, correctness by choice)

Uniform Destruction/Finalization

- Every type can have a destructor, ~T():
  - Non-trivial destructor == IDispose. Implicitly run when:
    - A stack based object goes out of scope
    - A class member's enclosing object is destroyed
    - A delete is performed on a pointer or handle. Example:

```
Object^ o = f();
delete o;  // run destructor now, collect memory later
```

- Every type can have a finalizer, !T():
  - The finalizer is executed at the usual times and subject to
    the usual guarantees, if the destructor has not already run

Deterministic Cleanup In C++

- C++ example:

```
void Transfer() {
  MessageQueue source( "server\\sourceQueue" );
  String^ qname = (String^)source.Receive().Body;
  MessageQueue      dest1( "server\\" + qname ),
                    dest2( "backup\\" + qname );
  Message^ message = source.Receive();
  dest1.Send( message );
  dest2.Send( message );
}
```

  - On exit (return or exception) from Transfer, destructible/ disposable objects have Dispose implicitly called in reverse order of construction. Here: dest2, dest1, and source
  - No finalization

Deterministic Cleanup In C#

- Minimal C# equivalent:

```
void Transfer() {
  using( MessageQueue source
           = new MessageQueue( "server\\sourceQueue" )
) {
    String qname = (String)source.Receive().Body;
    using( MessageQueue
        dest1 = new MessageQueue( "server\\" +
qname ),
        dest2 = new MessageQueue( "backup\\" + qname
) ) {
      Message message = source.Receive();
      dest1.Send( message );
      dest2.Send( message );
    }
  }
}
```

Deterministic Cleanup In VB/Java

- Minimal VB/Java equivalent (in C# syntax):

```
void Transfer() {
  MessageQueue source = null, dest1 = null, dest2 =
null;
  try {
    source = new MessageQueue( "server\\
sourceQueue" );
    String qname = (String)source.Receive().Body;
    dest1 = new MessageQueue( "server\\" + qname );
    dest2 = new MessageQueue( "backup\\" + qname );
    Message message = source.Receive();
    dest1.Send( message );
    dest2.Send( message );
  }
  finally {
    if( dest2 != null ) { dest2.Dispose(); }
    if( dest1 != null ) { dest1.Dispose(); }
    if( source != null ) { source.Dispose(); }
  }
```

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Generics × Templates

- Both are supported, and can be used together
- Generics:
  - Run-time, cross-language, and cross-assembly
  - Constraint based, less flexible than templates
  - Will eventually support many template features
- Templates:
  - Compile-time, C++, and generally intra-assembly (a template and its specializations in one assembly will also be available to friend assemblies)
  - Intra-assembly is not a high burden because you can

    expose templates through generic interfaces (e.g., expose a_container<T> via IList<T>)
  - Supports specialization, unique power programming

Generics

- Generics are declared much like templates:

```
generic<typename T>
where T : IDisposable, IFoo
ref class GR { // ...
  void f() {
    T t;
    t.Foo();
  } // call t.~T() implicitly
};
```

  - Constraints are inheritance-based

- Using generics and templates together works

  - Example: Generics can match template template params

```
template< template<class> class V >   // a TTP
void f() { V<int> v; /*...use v...*/ }
```

STL On The CLR

- C++ enables STL on CLR:
  - Verifiable
  - Separation of collections and algorithms
- Interoperates with Frameworks library
- C++ “for_each” and C# “for each” both work:

```
stdcli::vector<String^> v;

for_each( v.begin(), v.end(), functor );
for_each( v.begin(), v.end(), _1 += “suffix” );   // C++
for_each( v.begin(), v.end(), cout << _1 );       // lambdas

g( %v );            // signature of  g is g( IList<String^>^ )
```

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

CLR Types In The Native World

- Basic interop example:

```
class Data {
  XmlDocument* xmlDoc;
public:
  void Load( std::string fileName )  {
    XmlTextReader^ reader = gcnew XmlTextReader(

marshal_as<String^>( fileName ) );
    xmlDoc = new XmlDocument( reader );
  }
};
```

CLR Types In The Native World (2)

- Template<Ref> example:

```
template<class T>
void AFunctionTemplate( T ) { /*...*/ };

ref class Ref { /*...*/ };

Ref ref;
AFunctionTemplate( ref );          // ok
```

- Of course, any type can be templated:

```
template<class T>
ref class ARefTemplate { /*...*/ };        // ok
```

Native Types In The CLR World

- Basic interop example:

```
ref class MyControl : UserControl { //...  // reference type
  std::vector<std::string>* words;         // use native type
public:
  void Add( String^ s )
{ Add( marshal_as<std::string>(s)); }
  void Add( std::string s ) { words->push_back(s); }
};
```

What Customers Are Doing

- Example 1: Quake 2 extension example (using v1 syntax):

```
private __gc class RadarForm
  : public System::Windows::Forms::Form
{
  std::vector<RadarItem>* m_items;

public:
  RadarForm() : m_items( new
std::vector<RadarItem> )
    { /*...*/ };

  ~RadarForm() { delete items; }          // v1 finalizer
syntax

  // ... etc.
};
```

- Their first attempt was without the * (embed the

What Customers Are Doing (2)

- Example 2: Faking up base classes (e.g., expose native types to a CLR world)

```
private __gc class C {  // can't inherit from Native, so...
  Native* n;
public:
  C() : n( new Native ) { /*...*/ };
  ~C() { delete n; }
  void Foo( /*... a param list ...*/ )     { n->Foo( /*... */ ); }
  void Bar( /*... a param list ...*/ )     { n->Bar( /*... */ ); }
  // etc.
};
```

Future: Unified Type System, Object Model

- Arbitrary combinations of members and bases:
  - Any type can contain members and/or base classes of any other type. Virtual dispatch etc. work as expected
    - At most one base class may be of ref/value/mixed type
  - Overhead (regardless of mixing complexity, including deep inheritance with mixing & virtual overriding at each level):
    - For each object: At most one additional object
    - For each virtual function call: At most one additional virtual function call
- Pure type:
  - The declared type category, members, and bases are either all CLR, or all native
- Mixed type:
  - Everything else. Examples:

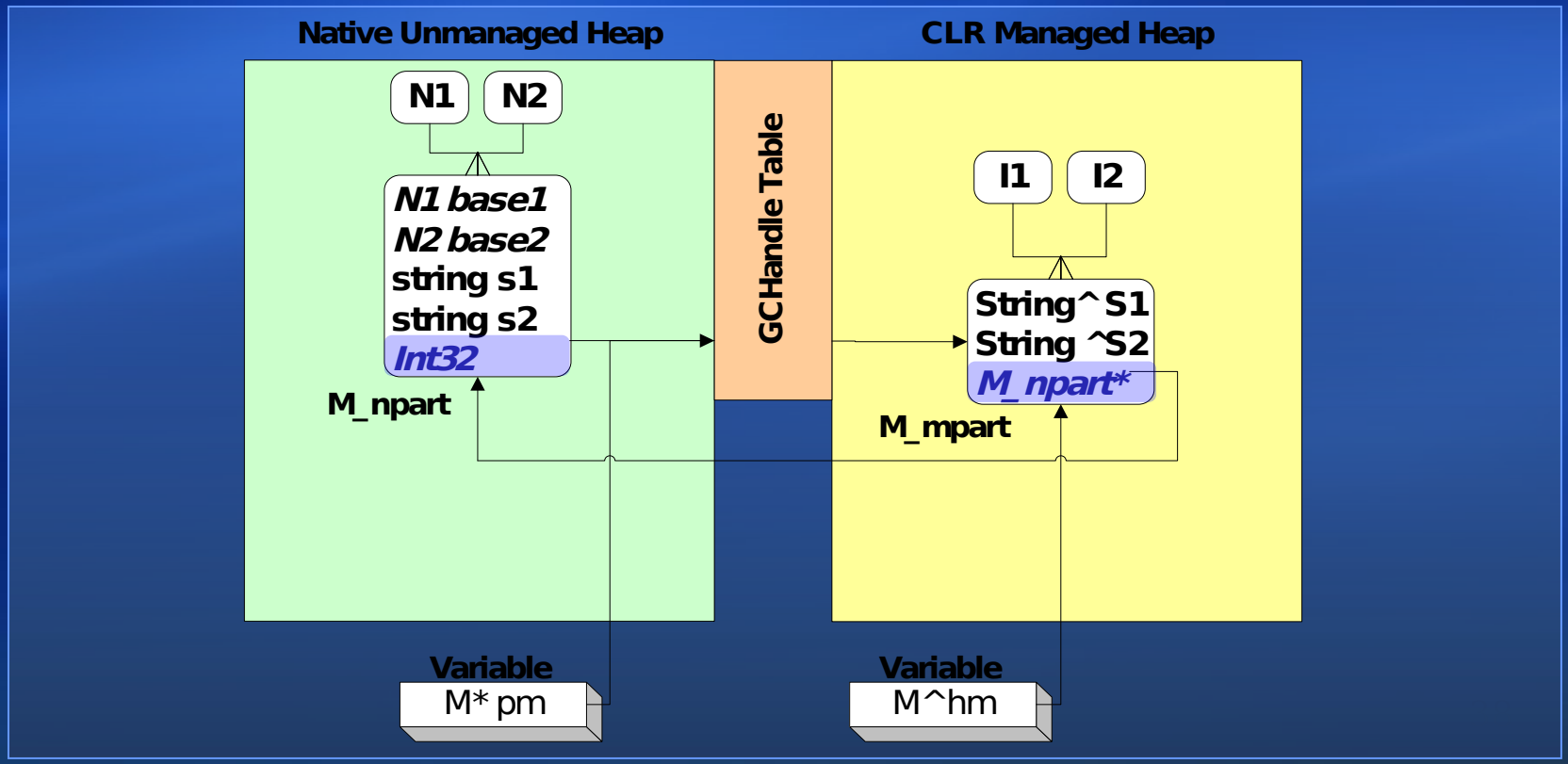
```
ref class Ref : R, public N1, N2 { string s; };
```

Future: Implementing Mixed Types

- 1 mixed = 1 pure + 1 pure

```
ref class M : I1, I2, N1, N2 {
  System::String ^S1, ^S2;
  std::string s1, s2;
};
```

```
M* pm = new M;
M^ hm = gcnew M;
```



Future: Result For Customer Code

V1 Syntax:

```
private __gc class RadarForm :
  public
System::Windows::Forms::Form
{
  std::vector<RadarItem>*
m_items;
  Native* n;

public:
  RadarForm() :
    : n( new Native )
    , m_items( new
std::vector<RadarItem> )
    { /*...*/ };

  ~RadarForm() { delete items;
delete n; }

  void Foo( /*... params ...*/ )
    { n->Foo( /*...*/ ); }

  void Bar( /*... params ...*/ )
```

V2 Syntax:

```
ref class RadarForm
  : System::Windows::Forms::Form
  , public Native
{
  std::vector<RadarItem> m_items;
};
```

One safe automated allocation, vs. N fragile handwritten allocations

This class is also better because it also has a destructor (implements IDisposable). That makes it work well by default with C++ automatic stack semantics (and C# using blocks)

Other Features

- Param arrays:
  - Created when needed, preferred over varargs

```
void f( String^ str, ... array<Object^>^ arr );
```

- Unified CLR and C++ operators:
  - Operators can now be static. Most work on handles

```
ref class R { public: // ...
  static R^ operator+( R^ lhs, R^ rhs );
};
```

  - Equality tests reference identity. Can be overridden by user
- Delegating constructors
- XML doc comments

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
   - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
   - Generics × templates, STL on CLR
   - Mixing native/CLR, other features
4. C++/CLI Standardization
   - Venue, players, timelines

Why Standardize C++/CLI?

- Primary motivators for C++/CLI standard:
  - Stability of language
  - C++ community understands and demands standards
  - Openness promotes adoption
  - Competing implementations should interoperate
- Same TC39, new TG5: C++/CLI
  - ISO C++ is still important to Visual C++. We remain actively involved in ISO WG21 and tracking ISO C++
  - C++/CLI is a binding between C++ and CLI only

C++/CLI: People And Timeline

- Who's who:
  - Convener: Tom Plum
  - Project Editor: Rex Jaeschke
  - Participants: Dinkumware, EDG, IBM, HP, Plum Hall
  - Independent conformance test suite: Plum Hall
- ECMA + ISO process:
  - October 1, 2003: ECMA TC39 plenary. Kick off TG5
  - December 2004: Adopt ECMA standard
  - December 2004: Kick off ISO fast-track process
  - December 2005: Adopt ISO standard

Overview

- 1. Rationale and Goals
- 2. Language Tour
- 3. Design and Implementation Highlights
  - Unifications: Type system, pointer and storage system (stack, native heap, gc heap), boxing
  - Deterministic cleanup: Destruction/Dispose, finalization, copying/assignment
  - Generics × templates, STL on CLR
  - Mixing native/CLR, other features
- 4. C++/CLI Standardization
  - Venue, players, timelines

Summary: C++ × CLR

- C++ features:
  - Deterministic cleanup, destructors
  - Templates
  - Native types
  - Multiple inheritance
  - STL, generic algorithms, lambda expressions
  - Pointer/pointee distinction (referring into boxes)
  - Copy construction, assignment
- CLR features:
  - Garbage collection, finalizers
  - Generics
  - CLR types
  - Interfaces
  - Verifiability
  - Security
  - Properties, delegates, events

The Two FAQs

- Q: Why should a C++ programmer use .NET?
  - GC, reflection, serialization, XML, etc. for existing projects
  - Full and easy access to the rich platform
- Q: Why should a .NET programmer use C++?
  - Easiest migration for existing code base: "Just use /clr."
  - Deterministic (and automatic) cleanup as usual in C++,
    no coding patterns
  - Easiest and most efficient native interop, incl. mixed types

Community Resources
Get Your Questions Answered!

- Newsgroups:
  - microsoft.public.dotnet.languages.vc
- Lounge: 309 Foyer
  - connect with Microsoft product teams, and PDC 2003 Speakers
- Ask The Experts:
  - Tuesday 7 pm – 9 pm in Hall G,H
- Web Sites:
  - http://pdcbloggers.net
  - http://msdn.microsoft.com/pdc/

Make the connection

Microsoft Professional Developers Conference 2003

October 26 - 30, 2003, Los Angeles, CA

Microsoft